School of Arts and Sciences

Department of Computer Science and Mathematics

PROXINA

Abdul Rahman Al Zaatari – 202201380 – 81906611

Ikram Bichbich 202208655

Rebecca Yazbeck - 202201494 - 79193508

CSC 430: Computer Networks

Dr. Ayman Tajjeddine

December 2nd, 2024

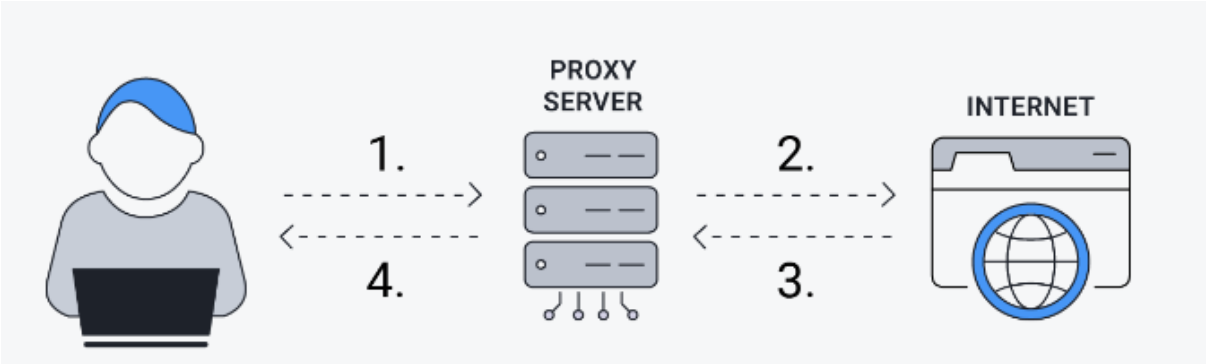
1 - Introduction 3
2 - Environment Setup 3
3 - Project Functionality 4
4 - Admin Interface 12
5 - Challenges Faced During Development 19
6 - Testing 20
Testing with apache 25
7 - Full List of Features in Bullet Points (so that Dr. Ayman does not get tired) 28
8 - Sources 30

1 - Introduction

This report introduces our **Proxy Server Application**, a client-server solution designed to mediate HTTP and HTTPS traffic efficiently. The application supports request filtering, caching, and real-time connection monitoring to enhance security and performance.

A web-based **Admin Interface**, built using Flask, allows administrators to manage blacklists, whitelists, logs, and cache entries. Secure login ensures authorized access, and a modern design provides a user-friendly experience.

Throughout the report, we outline the system's architecture, implementation, and the challenges encountered during development, alongside their solutions, highlighting our development journey!



2 - Environment Setup

The project was developed using **Visual Studio Code** (VSCode) as the Integrated Development Environment (IDE) and **Python 3.8** for scripting. Key libraries used include **socket**, **threading**, **sqlite3**, **ssl**, and **logging**, among others. The VSCode Python extension was configured for seamless code execution and debugging.

3 - Project Functionality

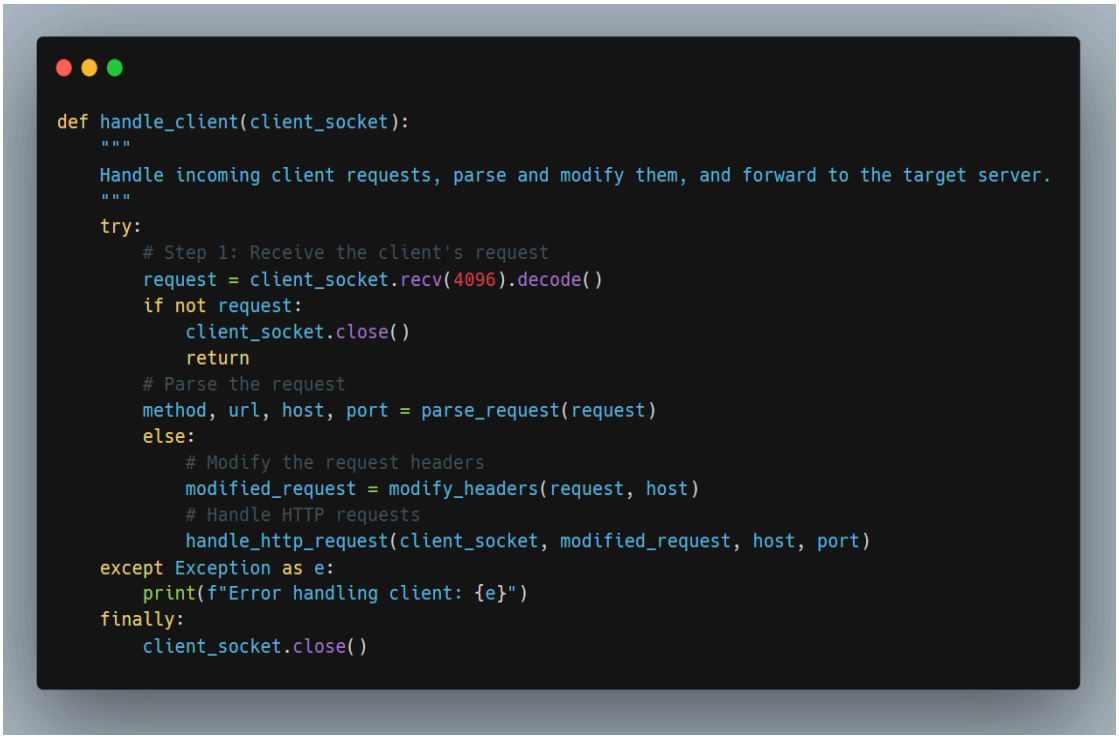
NOTE: Some of the implementations shown below were not final and have changed to accommodate new features! The section aims to show the core logic and functionalities.

This section explains how the proxy server achieves its core functionalities, the logic behind each implemented function, and the challenges encountered during development. Code snippets are provided for clarity. The section concludes with an overview of the changes made in the second version, discussing the motivations and challenges behind those changes.

1. Accepting and Forwarding Client Requests:

Logic: The proxy server accepts incoming client requests, parses them to extract necessary details (e.g., method, URL, host, and port), modifies headers for compatibility, and forwards the requests to the target server. This functionality ensures the proxy can handle requests effectively while maintaining compatibility with the target server.

Implementation:

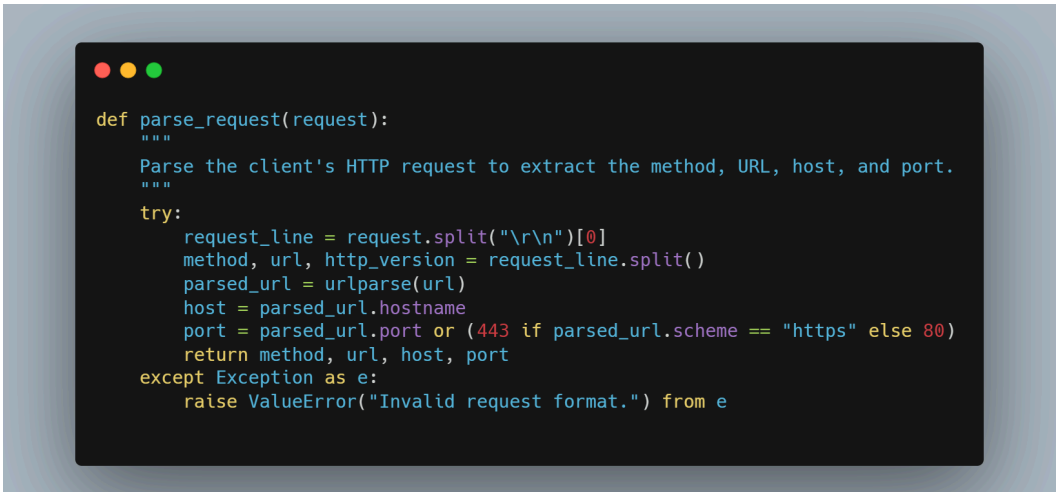


```python
def handle_client(client_socket):
    """
    Handle incoming client requests, parse and modify them, and forward to the target server.
    """
    try:
        # Step 1: Receive the client's request
        request = client_socket.recv(4096).decode()
        if not request:
            client_socket.close()
            return
        # Parse the request
        method, url, host, port = parse_request(request)
        else:
            # Modify the request headers
            modified_request = modify_headers(request, host)
            # Handle HTTP requests
            handle_http_request(client_socket, modified_request, host, port)
    except Exception as e:
        print(f"Error handling client: {e}")
    finally:
        client_socket.close()
```

2. Parsing Client Requests:

Logic: Parsing is critical to extract the HTTP method, URL, host, and port. This data determines where to forward the request.

Implementation:

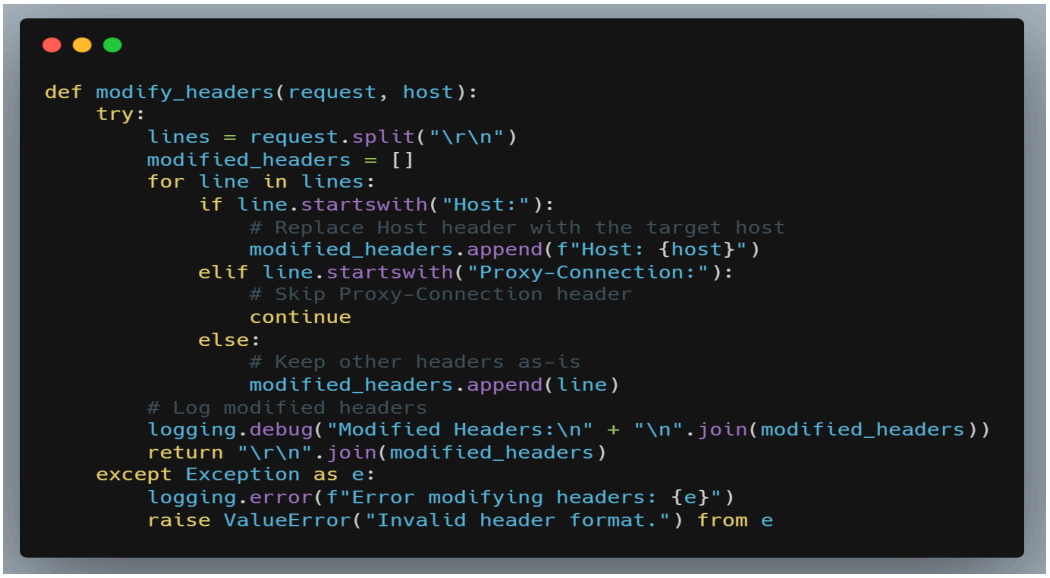


```python
def parse_request(request):
    """
    Parse the client's HTTP request to extract the method, URL, host, and port.
    """
    try:
        request_line = request.split("\r\n")[0]
        method, url, http_version = request_line.split()
        parsed_url = urlparse(url)
        host = parsed_url.hostname
        port = parsed_url.port or (443 if parsed_url.scheme == "https" else 80)
        return method, url, host, port
    except Exception as e:
        raise ValueError("Invalid request format.") from e
```

3. Modifying Headers:

Logic: Headers are modified to ensure compatibility with the target server by updating the Host header and removing proxy-specific headers like Proxy-Connection.

Implementation:



```python
def modify_headers(request, host):
    try:
        lines = request.split("\r\n")
        modified_headers = []
        for line in lines:
            if line.startswith("Host:"):
                # Replace Host header with the target host
                modified_headers.append(f"Host: {host}")
            elif line.startswith("Proxy-Connection:"):
                # Skip Proxy-Connection header
                continue
            else:
                # Keep other headers as-is
                modified_headers.append(line)
        # Log modified headers
        logging.debug("Modified Headers:\n" + "\n".join(modified_headers))
        return "\r\n".join(modified_headers)
    except Exception as e:
        logging.error(f"Error modifying headers: {e}")
        raise ValueError("Invalid header format.") from e
```

4. Forwarding Data:

Logic: Data is forwarded between the client and the target server using non-blocking I/O. This ensures efficient handling of requests and responses.

Implementation:

```python
def forward_data(client_socket, target_socket):
    sockets = [client_socket, target_socket]
    try:
        while True:
            ready_sockets, _, _ = select.select(sockets, [], [])
            for sock in ready_sockets:
                data = sock.recv(4096)
                if not data:
                    log_message("Connection closed.")
                    return

                if sock is client_socket:
                    target_socket.sendall(data)
                else:
                    client_socket.sendall(data)
    except Exception as e:
        log_message(f"Error during data forwarding: {e}")
    finally:
        client_socket.close()
        target_socket.close()
```

6. Caching and cleanup through expiry:

Caching is implemented to store server responses temporarily, reducing repeated server requests and improving the proxy's efficiency.

Logic: When a response is received, it is stored both in memory (using a cache dictionary) and in the database (proxy_data.db) within the cache table, with each entry assigned an expiration timestamp. Before forwarding the request, the proxy checks if the requested URL exists in the cache and is still valid; if so, the cached response is directly sent to the client, saving time and resources. To maintain cache relevance, a cleanup process periodically removed expired entries from both the in-memory cache and the database, ensuring only the up-to-date responses are available.

Implementation:

```python
def get_cached_response(url):
    with cache_lock:
        if url in cache:
            entry = cache[url]
            if entry["expiry"] > time.time():
                log_message(f"Cache hit for URL: {url}")
                return entry["data"]
            else:
                log_message(f"Cache expired for URL: {url}")
                del cache[url]
    return None
```

```python
def cache_response(url, response_data, headers):
    with cache_lock:
        expiry = time.time() + 60  # Default expiry time (60 seconds)
        # Update in-memory cache
        cache[url] = {
            "data": response_data,
            "expiry": expiry
        }
        # Log cache addition
        log_message(f"Cached response for {url}. Expires at: {expiry}")
        # Store the cache entry in the SQLite database
        try:
            conn = sqlite3.connect("proxy_data.db")
            cursor = conn.cursor()
            # Insert or replace the entry into the cache table
            cursor.execute("""
                INSERT OR REPLACE INTO cache (url, data, expiry)
                VALUES (?, ?, ?)
            """, (url, response_data, datetime.fromtimestamp(expiry)))
            conn.commit()
            log_message(f"Cache entry added to database for {url}")
        except sqlite3.Error as e:
            log_message(f"Error saving cache entry to database: {e}")
        finally:
            conn.close()
```

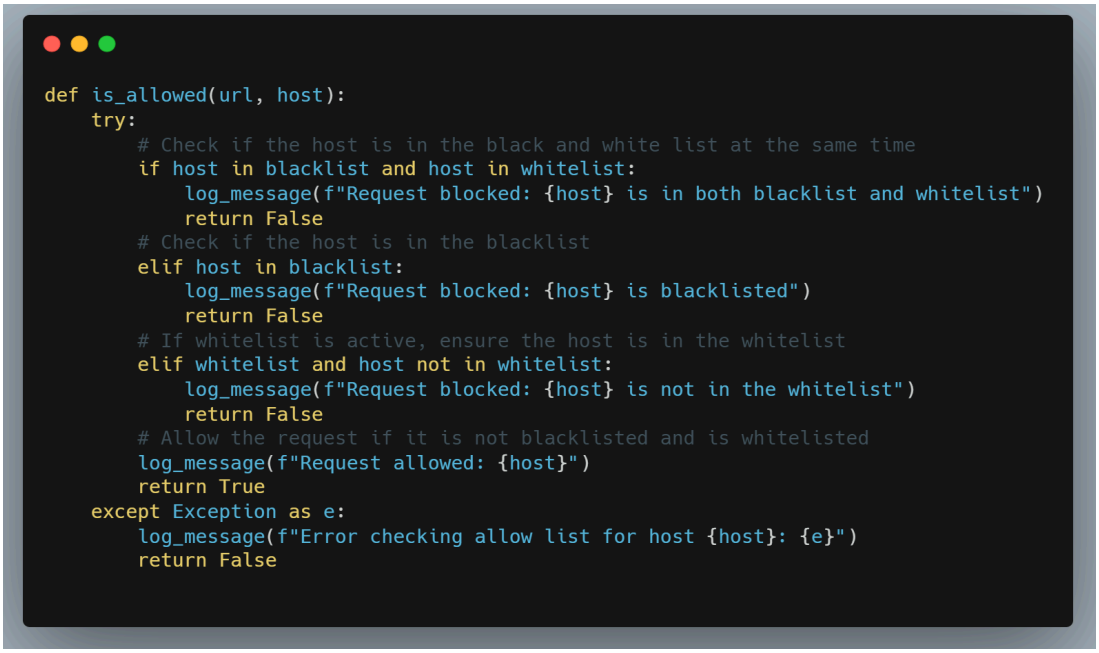
```python
def cleanup_cache():
    with cache_lock:
        current_time = time.time()
        expired_urls = [url for url, entry in cache.items() if entry["expiry"] <= current_time]

        # Remove expired entries from in-memory cache
        for url in expired_urls:
            del cache[url]
            log_message(f"Cache entry expired and removed: {url}")

        # Remove expired entries from the SQLite database
        try:
            conn = sqlite3.connect("proxy_data.db")
            cursor = conn.cursor()
            cursor.execute("DELETE FROM cache WHERE expiry <= ?",
(datetime.fromtimestamp(current_time),))
            conn.commit()
            #log_message("Expired cache entries removed from database.")
        except sqlite3.Error as e:
            log_message(f"Error cleaning up cache entries in database: {e}")
        finally:
            conn.close()
```

6. Whitelist/Blacklist:

This feature restricts or allows access to certain domains.

Logic: These lists are dynamically loaded from the **blacklist** and **whitelist** tables in the **proxy_data.db** database. Requests are validated by checking whether the domain is in the blacklist (requests are blocked) or the whitelist (only listed domains are allowed if the whitelist is active). To ensure the lists stay updated, a dedicated thread (**refresh_lists**) periodically refreshes them every 60 seconds. A domain cannot be in both the whitelist or blacklist at the same time.

Implementation:



```python
def is_allowed(url, host):
    try:
        # Check if the host is in the black and white list at the same time
        if host in blacklist and host in whitelist:
            log_message(f"Request blocked: {host} is in both blacklist and whitelist")
            return False
        # Check if the host is in the blacklist
        elif host in blacklist:
            log_message(f"Request blocked: {host} is blacklisted")
            return False
        # If whitelist is active, ensure the host is in the whitelist
        elif whitelist and host not in whitelist:
            log_message(f"Request blocked: {host} is not in the whitelist")
            return False
        # Allow the request if it is not blacklisted and is whitelisted
        log_message(f"Request allowed: {host}")
        return True
    except Exception as e:
        log_message(f"Error checking allow list for host {host}: {e}")
        return False
```

7. Handling Http Request:

Logic: The proxy server manages HTTP requests by forwarding them to the target server and caching the responses for optimized performance. The server first checks whether the requested URL's response is already cached. If cached data is available, it is sent directly to the client, ensuring faster response times. Otherwise, the proxy establishes a connection with the target server, forwards the client's request, and reads the server's response. The response headers and body are processed and cached for future requests. This mechanism ensures efficient handling of HTTP requests while reducing server load and response time.

Implementation:

```
def handle_http_request(client_socket, request, host, port):
    try:
        # Construct URL and check cache
        url = f"http://{host}{urlparse(request.split(' ')[1]).path}"
        cached_data = get_cached_response(url)
        if cached_data:
            client_socket.sendall(cached_data)  # Serve from cache
            return

        # Forward request to target server
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as target_socket:
            target_socket.connect((host, port))
            target_socket.sendall(request.encode())
            # Receive and cache response
            response_data = target_socket.recv(4096)
            cache_response(url, response_data)
            client_socket.sendall(response_data)  # Serve to client
    except Exception as e:
        log_message(f"Error: {e}")
        client_socket.sendall(b"HTTP/1.1 500 Internal Server Error\r\n\r\n")
    finally:
        client_socket.close()
```

8. Extra scheduled cleanup:

Logic: The extra scheduled cleanup process is responsible for periodically removing expired entries from the cache to ensure efficient memory usage and data integrity. This cleanup process runs in the background, checking for and deleting stale cache entries based on a predefined time interval (e.g., every 24 hours). For testing, a shorter interval (e.g., 10 seconds) was used. The functionality also gracefully handles any errors during execution and logs relevant information.

Implementation:

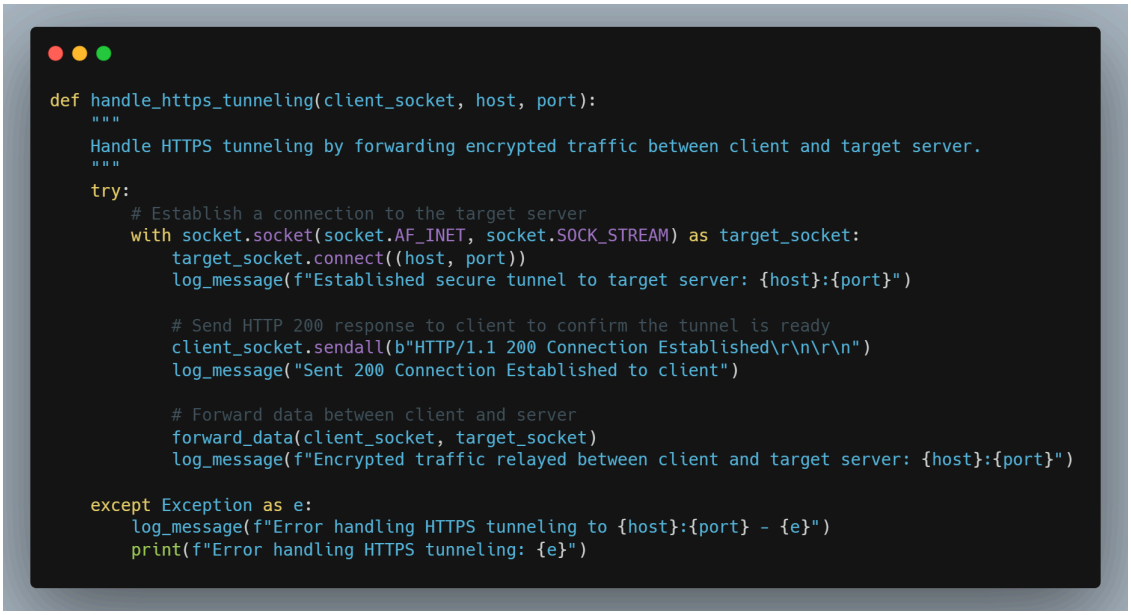
```
def schedule_cleanup():
    try:
        while not stop_event.is_set():
            time.sleep(1*60)  # Change to 24 * 60 * 60 for daily cleanup
            cleanup_cache()
    except Exception as e:
        log_message(f"Error in scheduled cache cleanup: {e}")
    finally:
        log_message("Scheduled cache cleanup stopped.")
```

9. Handling HTTPS:

Logic: The proxy server facilitates HTTPS tunneling by creating a secure connection between the client and the target server, allowing encrypted traffic to flow seamlessly. Upon receiving a CONNECT request, the server first establishes a connection with the target server using the provided host and port. Once the connection is successful, the server sends an HTTP 200 **Connection Established** response to the client, confirming that the tunnel is ready.

The proxy then enters a bidirectional data-forwarding mode, relaying encrypted traffic between the client and the target server without decrypting it. This process ensures secure and efficient communication for HTTPS requests. In case of errors, such as connection failures or data transmission issues, detailed logs are generated to assist in diagnosing and resolving the problem. This mechanism ensures robust handling of HTTPS connections while maintaining security and reliability.

Implementation



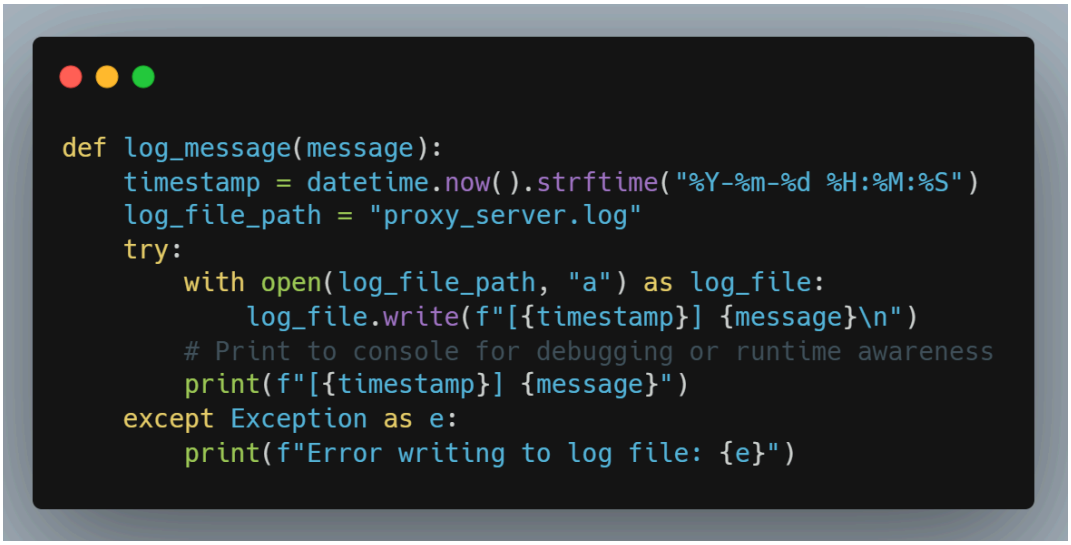
```python
def handle_https_tunneling(client_socket, host, port):
    """
    Handle HTTPS tunneling by forwarding encrypted traffic between client and target server.
    """
    try:
        # Establish a connection to the target server
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as target_socket:
            target_socket.connect((host, port))
            log_message(f"Established secure tunnel to target server: {host}:{port}")

            # Send HTTP 200 response to client to confirm the tunnel is ready
            client_socket.sendall(b"HTTP/1.1 200 Connection Established\r\n\r\n")
            log_message("Sent 200 Connection Established to client")

            # Forward data between client and server
            forward_data(client_socket, target_socket)
            log_message(f"Encrypted traffic relayed between client and target server: {host}:{port}")

    except Exception as e:
        log_message(f"Error handling HTTPS tunneling to {host}:{port} - {e}")
        print(f"Error handling HTTPS tunneling: {e}")
```

10- Logging and debugging:

The logging and debugging goal in the proxy is to provide detailed insights about its internal and external operations.

Logic: Whenever we have events such as connection errors, cache hits or misses, and list refreshes, they will be logged in the **proxy_server.log** file. Additionally, log messages were printed to the console for testing purposes, allowing for real-time monitoring and debugging of the proxy's performance and behavior.

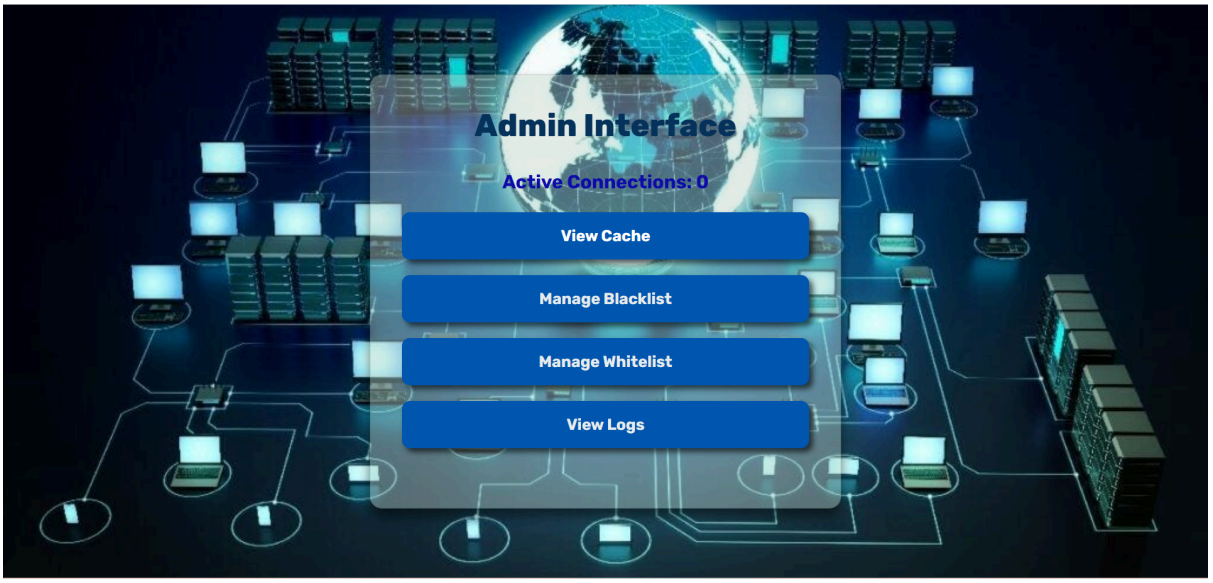
Implementation:



```python
def log_message(message):
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    log_file_path = "proxy_server.log"
    try:
        with open(log_file_path, "a") as log_file:
            log_file.write(f"[{timestamp}] {message}\n")
        # Print to console for debugging or runtime awareness
        print(f"[{timestamp}] {message}")
    except Exception as e:
        print(f"Error writing to log file: {e}")
```

4 - Admin Interface

The Admin Interface is designed to allow administrators to efficiently manage various aspects of the application, such as blacklists, whitelists, logs, cache, and monitoring active connections. The interface is secured by a login system, ensuring that only authenticated administrators can access it.



Key Features of the Admin Interface:

1. Login System:

- Admins log in using an email and password stored in the database.
- The **session** is used to track login state, ensuring that only logged-in users can access the admin interface.

Implementation:

```python
@app.route("/admin/login", methods=["GET", "POST"])
def admin_login():
    """Admin login page."""
    if request.method == "POST":
        email = request.form["email"]
        password = request.form["password"]
        # Check if the email and password match an admin in the database
        query = "SELECT * FROM admin WHERE email = ? AND password = ?"
        result = query_db(query, (email, password), one=True)
        if result:
            # Store the email in session to keep track of logged-in user
            session["admin_logged_in"] = True
            session["admin_email"] = email
            return redirect(url_for("index"))
        else:
            flash("Invalid credentials. Please try again.", "danger")
            return redirect(url_for("admin_login"))

    return render_template("login.html")
```

2. Dashboard:

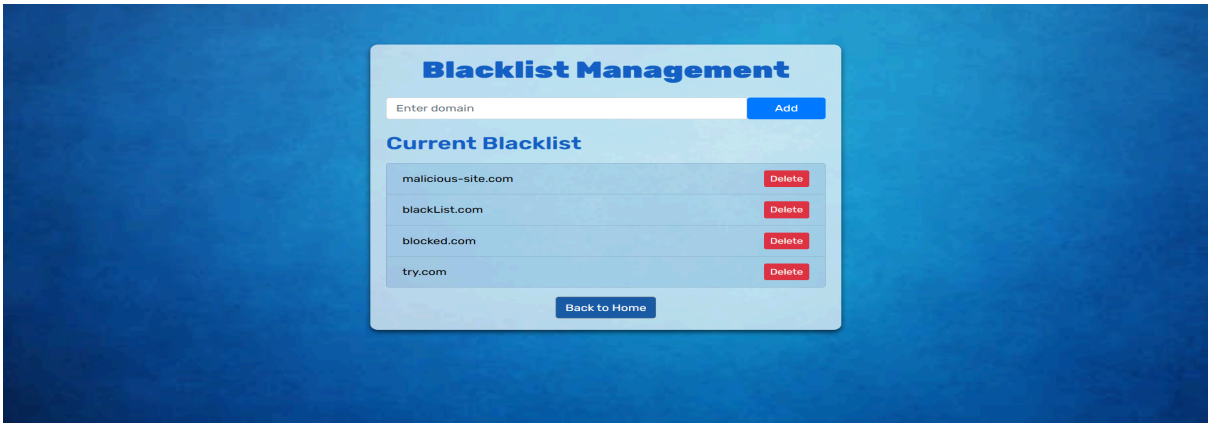
- After logging in, the admin is redirected to the main dashboard, which serves as the central interface for managing the system.

Implementation:

```python
@app.route("/")
def index():
    """Render the main admin interface."""
    if "admin_logged_in" not in session or not session["admin_logged_in"]:
        return redirect(url_for("admin_login"))
    return render_template("index.html")
```

3. Blacklist Management:

- Admins can add or remove domains from a blacklist.
- These domains are likely blocked or flagged for suspicious activity.



Implementation:

```python
@app.route("/blacklist", methods=["GET", "POST"])
def blacklist():
    """Manage the blacklist."""
    if request.method == "POST":
        domain = request.form["domain"]
        query_db("INSERT OR IGNORE INTO blacklist (domain) VALUES (?)",
(domain,))
        return redirect(url_for("blacklist"))

    blacklist = query_db("SELECT domain FROM blacklist")
    return render_template("blacklist.html", blacklist=blacklist)

@app.route("/blacklist/delete/<domain>", methods=["POST"])
def delete_blacklist(domain):
    """Delete a domain from the blacklist."""
    query_db("DELETE FROM blacklist WHERE domain = ?", (domain,))
    return redirect(url_for("blacklist"))
```

4. Whitelist Management:

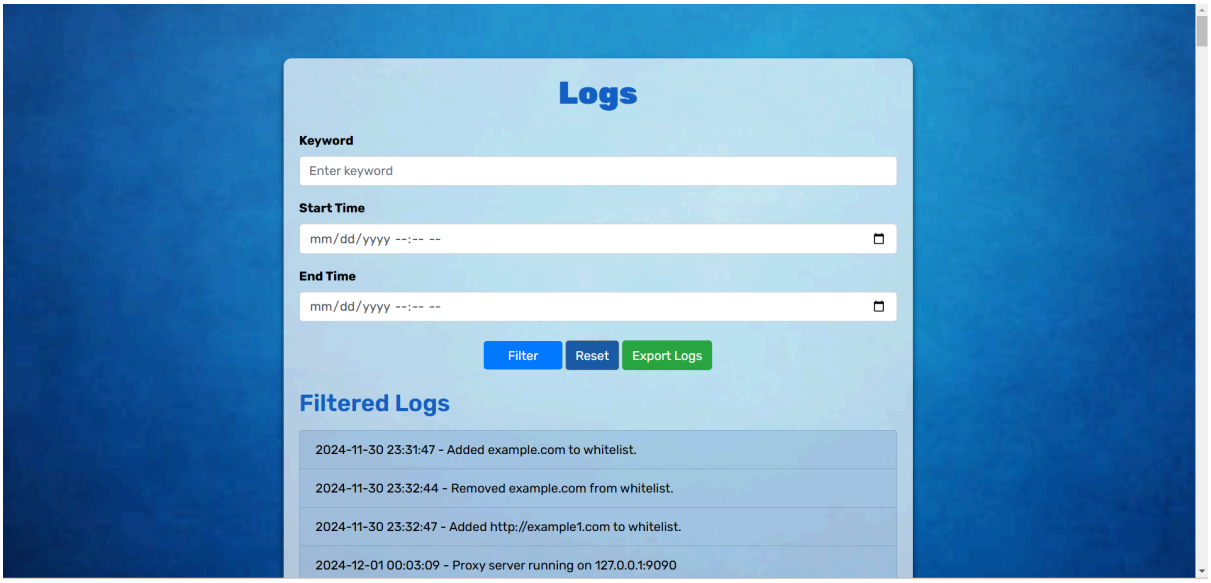
- Admins can add or remove domains from a whitelist.
- This allows explicitly permitting certain domains for use.

Implementation:

```
@app.route("/whitelist", methods=["GET", "POST"])
def whitelist():
    """Manage the whitelist."""
    if request.method == "POST":
        domain = request.form["domain"]
        query_db("INSERT OR IGNORE INTO whitelist (domain) VALUES (?)",
(domain,))
        return redirect(url_for("whitelist"))

    whitelist = query_db("SELECT domain FROM whitelist")
    return render_template("whitelist.html", whitelist=whitelist)

@app.route("/whitelist/delete/<domain>", methods=["POST"])
def delete_whitelist(domain):
    """Delete a domain from the whitelist."""
    query_db("DELETE FROM whitelist WHERE domain = ?", (domain,))
    return redirect(url_for("whitelist"))
```

6. Logs Management:

- Admins can view logs that detail system activities.
- Logs can be filtered by keyword and time range to pinpoint specific information.

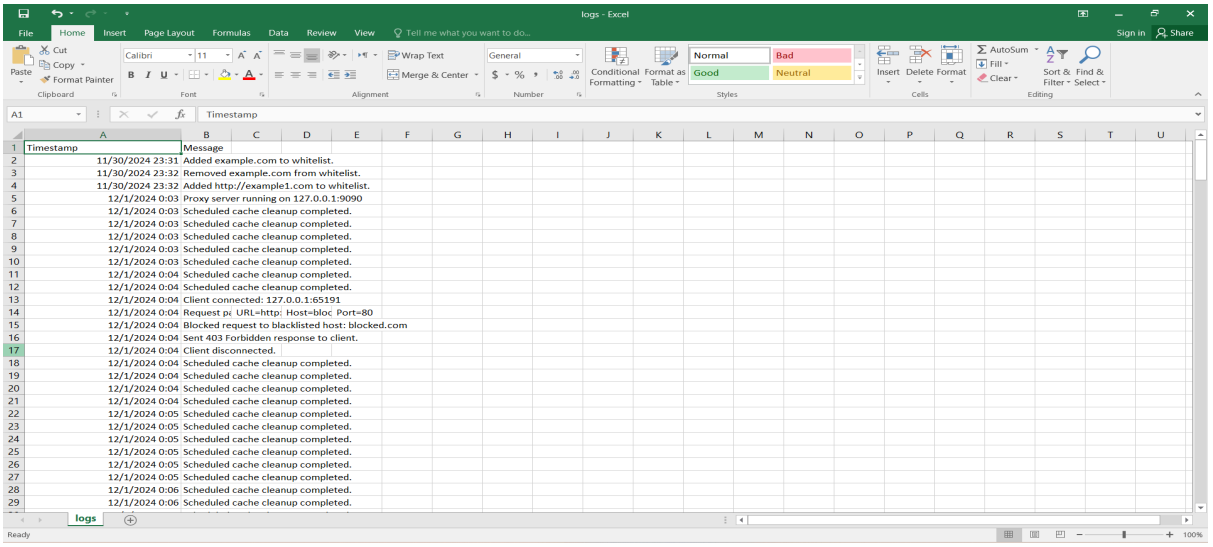


Implementation:

```python
@app.route("/logs", methods=["GET"])

def logs():
    """View and filter logs based on query parameters."""
    keyword = request.args.get("keyword", "")
    start_time = request.args.get("start_time", "")
    end_time = request.args.get("end_time", "")

    query = "SELECT timestamp, message FROM logs WHERE 1=1"
    params = []

    if keyword:
        query += " AND message LIKE ?"
        params.append(f"%{keyword}%")

    if start_time:
        try:
            start_time = datetime.strptime(start_time, "%Y-%m-%dT%H:%M").strftime("%Y-%m-%d %H:%M:%S")
            query += " AND timestamp >= ?"
            params.append(start_time)
        except ValueError as e:
            print(f"Start time formatting error: {e}")
    if end_time:
        try:
            end_time = datetime.strptime(end_time, "%Y-%m-%dT%H:%M").strftime("%Y-%m-%d %H:%M:%S")
            query += " AND timestamp <= ?"
            params.append(end_time)
        except ValueError as e:
            print(f"End time formatting error: {e}")

    print("SQL Query:", query)
    print("Parameters:", params)

    logs = query_db(query, params)
    return render_template(
        "logs.html",
        logs=logs,
        keyword=keyword,
        start_time=start_time,
        end_time=end_time
    )
```

6. Export Logs:

- Admins can export filtered logs to a CSV file for offline analysis or reporting purposes.

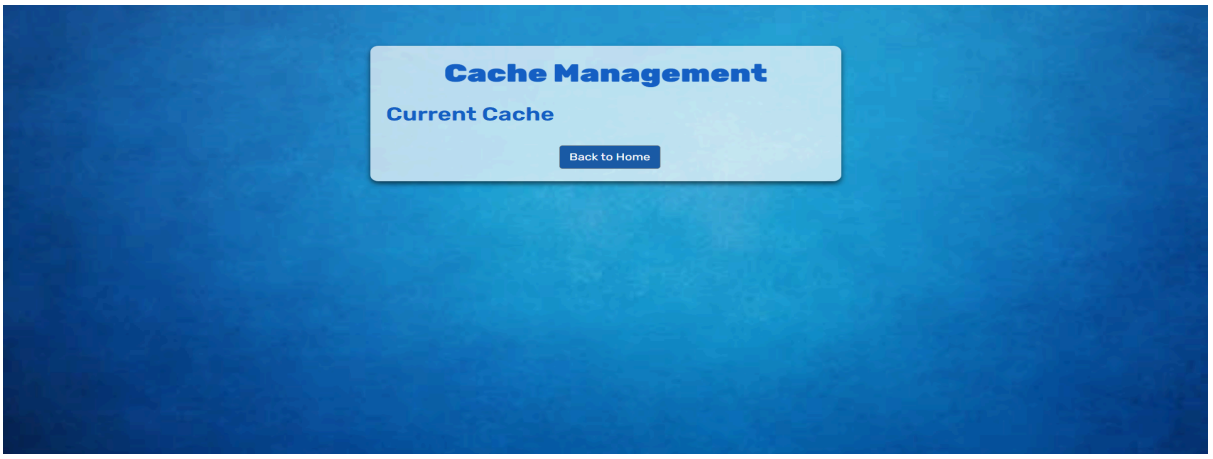


Implementation:

```python
@app.route("/logs/export", methods=["GET"])
def export_logs():
    """Export logs to a CSV file."""
    keyword = request.args.get("keyword", "")
    start_time = request.args.get("start_time", "")
    end_time = request.args.get("end_time", "")

    query = "SELECT timestamp, message FROM logs WHERE 1=1"
    params = []

    if keyword:
        query += " AND message LIKE ?"
        params.append(f"%{keyword}%")
    if start_time:
        try:
            start_time = datetime.strptime(start_time, "%Y-%m-%dT%H:%M").strftime("%Y-%m-%d %H:%M:%S")
            query += " AND timestamp >= ?"
            params.append(start_time)
        except ValueError as e:
            print(f"Start time formatting error: {e}")
    if end_time:
        try:
            end_time = datetime.strptime(end_time, "%Y-%m-%dT%H:%M").strftime("%Y-%m-%d %H:%M:%S")
            query += " AND timestamp <= ?"
            params.append(end_time)
        except ValueError as e:
            print(f"End time formatting error: {e}")

    logs = query_db(query, params)

    # Create CSV response
    def generate_csv():
        yield "Timestamp,Message\n"
        for log in logs:
            yield f"{log[0]},{log[1]}\n"

    return Response(
        generate_csv(),
        mimetype="text/csv",
        headers={"Content-Disposition": "attachment;filename=logs.csv"}
    )
```

7. Cache Management:

- Admins can view cache entries for optimized performance.



Implementation:

```python
@app.route("/cache", methods=["GET", "POST"])
def cache():
    """View and manage cache entries."""
    if request.method == "POST":
        # Add a new cache entry
        url = request.form["url"]
        data = request.form["data"]
        expiry = request.form["expiry"]
        query_db("INSERT OR REPLACE INTO cache (url, data, expiry) VALUES (?, ?, ?)", (url, data,
expiry))
        return redirect(url_for("cache"))

    # View all cache entries
    cache_entries = query_db("SELECT url, data, expiry FROM cache")
    return render_template("cache.html", cache=cache_entries)

@app.route("/cache/clear", methods=["POST"])
def clear_cache():
    """Clear cache entries."""
    url = request.form.get("url", None)
    if url:
        query_db("DELETE FROM cache WHERE url = ?", (url,))
    else:
        query_db("DELETE FROM cache")  # Clear all cache entries
    return redirect(url_for("cache"))
```

8. Active Connections:

- Displays the number of currently active connections, helping administrators monitor usage.

Implementation:

```python
@app.route("/active_connections")
    def get_active_connections():
        # Query the active connections from the settings table
        result = query_db("SELECT active_connections FROM settings LIMIT 1", one=True)
        return result[0] if result else 0  # Return 0 if no result is found
    try:
        active_connections = get_active_connections()  # Fetch the latest value from DB
        return jsonify({"active_connections": active_connections})
    except Exception as e:
        return jsonify({"error": str(e)}), 500
```

9. Logout Functionality:

- Securely logs out admins by clearing their session data.

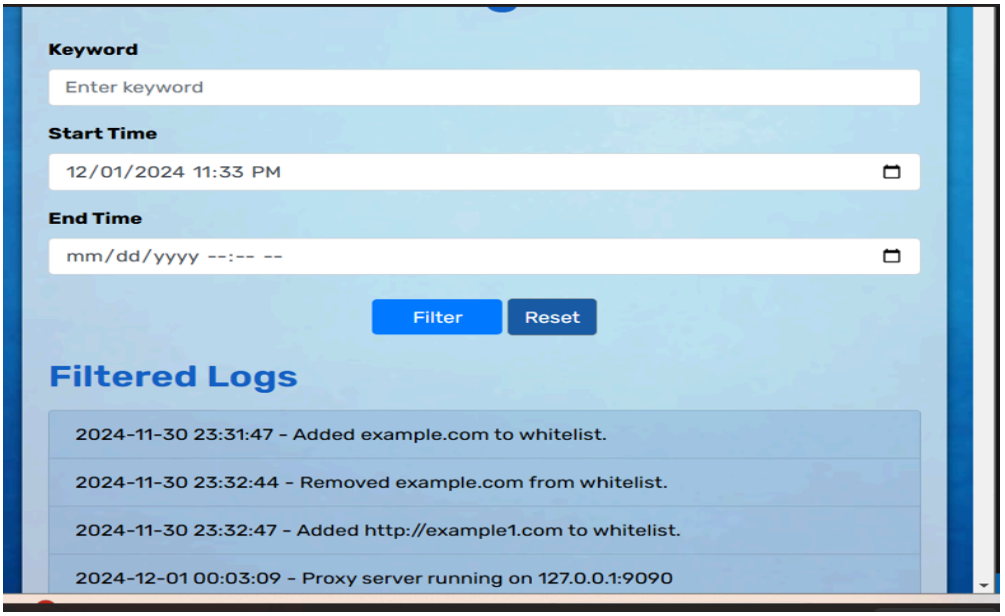
```python
@app.route("/active_connections")
def active_connections_view():
    """View the number of active connections."""

    def get_active_connections():
        # Query the active connections from the settings table
        result = query_db("SELECT active_connections FROM settings LIMIT 1", one=True)
        return result[0] if result else 0  # Return 0 if no result is found

    try:
        active_connections = get_active_connections()  # Fetch the latest value from DB
        return jsonify({"active_connections": active_connections})
    except Exception as e:
        return jsonify({"error": str(e)}), 500
```

5 - Challenges Faced During Development

1. **Time Formatting Issues in Logs**:

   Problem: Initially, the default logging system captured only hours and minutes, setting seconds to **00**. This caused mismatches when querying logs by precise time ranges.For example, the start time was supposed to filter and capture the data after **23:33:00**, but in this case, data from **23:31:47** was also captured in the filtered logs.

   Solution: Updated the logging function to include seconds, ensuring accurate timestamps in the format **YYYY-MM-DD HH:MM:SS**.



2. **CSV Export**:

   Problem: While implementing the CSV export feature, ensuring compatibility with the filtering logic was a challenge. The exported data initially ignored filters applied on the logs page.

   Solution: Dynamically pass query parameters (**keyword, start_time, end_time**) to the export route and ensure the generated CSV reflects the filtered data.

3. **Connecting the database for the admin interface**:

   Problem: Having to switch a lot of the proxy functionalities and have them stored inside the database or use the database was hard but needed for the admin interface.

   Solution: Reworked all of the proxy functions and inserted the database logic to act as a connection layer between the admin and the proxy.

4. The file was not being cached from the first request as the caching process was done after the forwarding of the data. We only had to modify the order of the code to solve this issue.

6 - Testing

All previous logs were deleted before proceeding with the tests.

6.1- Run the system and check the logs:

The system was launched in a dedicated Python terminal. The server was running successfully according to the log files.

proxy_server.log

```
1   [2024-12-08 13:02:15] Blacklist/Whitelist refresh thread started.
2   [2024-12-08 13:02:15] Cache cleanup thread started.
3   [2024-12-08 13:02:15] Active connections: 0
4   [2024-12-08 13:02:15] Active connections monitoring started.
5   [2024-12-08 13:02:15] Proxy server running on 127.0.0.1:9090
6
```

```
2024-12-08 13:02:15] Blacklist/Whitelist refresh thread started.
024-12-08 13:02:15,058 - INFO - Refreshed blacklist and whitelist.
2024-12-08 13:02:15] Cache cleanup thread started.
2024-12-08 13:02:15] Active connections monitoring started.
2024-12-08 13:02:15] Active connections: 0
roxy server running on 127.0.0.1:9090
2024-12-08 13:02:15] Proxy server running on 127.0.0.1:9090
```

6.2 - Testing with an HTTP request:

While executing the command curl.exe -x http://127.0.0.1:9090 http://example.com the content was successfully displayed on the separate terminal where the command was executed. The log file indicated that the connection was accepted for the HTTP request and that the file was being cached.

```
PS C:\Users\Hp\Desktop\UNI\Fall 2024\network\project\project> curl.exe -x http://127.0.0.1:9090 http://example.com
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 0;
        padding: 0;
        font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;

    }
    div {
        width: 600px;
        margin: 5em auto;
        padding: 2em;
        background-color: #fdfdff;
        border-radius: 0.5em;
        box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);
    }
    a:link, a:visited {
        color: #38488f;
        text-decoration: none;
    }
    @media (max-width: 700px) {
        div {
            margin: 0 auto;
            width: auto;
        }
    }
    </style>
</head>

<body>
<div>
    <h1>Example Domain</h1>
    <p>This domain is for use in illustrative examples in documents. You may use this
    domain in literature without prior coordination or asking for permission.</p>
    <p><a href="https://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
PS C:\Users\Hp\Desktop\UNI\Fall 2024\network\project\project>
```

Log file:

```
[2024-12-08 13:07:14] Accepted connection from ('127.0.0.1', 64829)
[2024-12-08 13:07:14] New connection. Active connections: 0
[2024-12-08 13:07:14] Client connected: 127.0.0.1:64829
[2024-12-08 13:07:14] Request parsed: Method=GET, URL=http://example.com/, Host=example.com, Port=80
[2024-12-08 13:07:14] Request allowed: example.com
[2024-12-08 13:07:14] Handling HTTP request for example.com:80
[2024-12-08 13:07:14] Connected to target server: example.com:80
[2024-12-08 13:07:14] Forwarded request to target server: example.com:80
[2024-12-08 13:07:15] Expired cache entries removed from database.
[2024-12-08 13:07:15] Received headers: HTTP/1.1 200 OK

Age: 231812

Cache-Control: max-age=604800

Content-Type: text/html; charset=UTF-8

Date: Sun, 08 Dec 2024 11:07:14 GMT

Etag: "3147526947+gzip+ident"

Expires: Sun, 15 Dec 2024 11:07:14 GMT

Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT

Server: ECAcc (dcd/7D77)

Vary: Accept-Encoding

X-Cache: HIT

Content-Length: 1256
```

6.3 - Testing an HTTP Post request:

After adding httpbin.org to the whitelist, the command curl.exe -x http://127.0.0.1:9090 -X POST -d '{"key":"value"}' -H "Content-Type: application/json" http://httpbin.org/post was successful.

```
PS C:\Users\Hp\Desktop\UNI\Fall 2024\network\project\project> curl.exe -x http://127.0.0.1:9090 -X POST -d '{"key":"value"}' -H "Content-Type: application/json" http://httpbin.org/post
{
  "args": {},
  "data": "{key:value}",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Content-Length": "11",
    "Content-Type": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "curl/8.9.1",
    "X-Amzn-Trace-Id": "Root=1-675588ff-270c080d033da2fd1d0c1dcd"
  },
  "json": null,
  "origin": "178.135.16.201",
  "url": "http://httpbin.org/post"
}
PS C:\Users\Hp\Desktop\UNI\Fall 2024\network\project\project>
```

The log file indicated the following:

```
[2024-12-08 13:54:39] Accepted connection from ('127.0.0.1', 65299)
[2024-12-08 13:54:39] New connection. Active connections: 0
[2024-12-08 13:54:39] Client connected: 127.0.0.1:65299
[2024-12-08 13:54:39] Request parsed: Method=POST, URL=http://httpbin.org/post, Host=httpbin.org, Port=80
[2024-12-08 13:54:39] Request allowed: httpbin.org
[2024-12-08 13:54:39] Handling HTTP request for httpbin.org:80
[2024-12-08 13:54:40] Connected to target server: httpbin.org:80
[2024-12-08 13:54:40] Forwarded request to target server: httpbin.org:80
[2024-12-08 13:54:41] Received headers: HTTP/1.1 200 OK

Date: Sun, 08 Dec 2024 11:54:40 GMT

Content-Type: application/json

Content-Length: 399

Connection: keep-alive

Server: gunicorn/19.9.0

Access-Control-Allow-Origin: *

Access-Control-Allow-Credentials: true
[2024-12-08 13:54:41] Cached response for http://httpbin.org/post. Expires at: 1733658941.4188256
[2024-12-08 13:54:41] Cache entry added to database for http://httpbin.org/post
[2024-12-08 13:55:07] Expired cache entries removed from database.
[2024-12-08 13:55:11] Timeout during receive from httpbin.org:80
[2024-12-08 13:55:11] Cached response for http://httpbin.org/post. Expires at: 1733658971.4630075
[2024-12-08 13:55:11] Cache entry added to database for http://httpbin.org/post
[2024-12-08 13:55:11] Response fully cached for http://httpbin.org/post
[2024-12-08 13:55:11] Client disconnected.
```

6.4 - Testing caching:

After executing the curl.exe -x http://127.0.0.1:9090 http://example.com twice within less than 1 minute, the display of the file was faster, and the log file indicated that it was a cache hit for the url: http://example.com.

```
[2024-12-08 14:03:48] Accepted connection from ('127.0.0.1', 65370)
[2024-12-08 14:03:48] New connection. Active connections: 1
[2024-12-08 14:03:48] Client connected: 127.0.0.1:65370
[2024-12-08 14:03:48] Request parsed: Method=GET, URL=http://example.com/, Host=example.com, Port=80
[2024-12-08 14:03:48] Request allowed: example.com
[2024-12-08 14:03:48] Handling HTTP request for example.com:80
[2024-12-08 14:03:48] Cache hit for URL: http://example.com/
[2024-12-08 14:03:48] Cache hit: http://example.com/
[2024-12-08 14:03:48] Client disconnected.
[2024-12-08 14:03:48] Connection closed. Active connections: 0
[2024-12-08 14:03:48] Expired cache entries removed from database.
```

6.5 - Testing blacklist:

After running curl.exe -x http://127.0.0.1:9090 http://try.com with try.com being blacklisted, we received the expected 403 Forbidden output with the log file indicating the following:

```
[2024-12-08 14:06:40] Accepted connection from ('127.0.0.1', 65403)
[2024-12-08 14:06:40] New connection. Active connections: 0
[2024-12-08 14:06:40] Client connected: 127.0.0.1:65403
[2024-12-08 14:06:40] Request parsed: Method=GET, URL=http://try.com/, Host=try.com, Port=80
[2024-12-08 14:06:40] Request blocked: try.com is blacklisted
[2024-12-08 14:06:40] Blocked request to try.com due to whitelist/blacklist restrictions.
[2024-12-08 14:06:40] Sent 403 Forbidden response to client
[2024-12-08 14:06:40] Client connection closed after rejection.
[2024-12-08 14:06:40] Connection closed. Active connections: -1
[2024-12-08 14:06:40] Expired cache entries removed from database.
```

After adding try.com to the whitelist and removing it from the blacklist and running the same command again:

```
[2024-12-08 14:10:14] Accepted connection from ('127.0.0.1', 65439)
[2024-12-08 14:10:14] New connection. Active connections: 0
[2024-12-08 14:10:14] Client connected: 127.0.0.1:65439
[2024-12-08 14:10:14] Request parsed: Method=GET, URL=http://try.com/, Host=try.com, Port=80
[2024-12-08 14:10:14] Request allowed: try.com
[2024-12-08 14:10:14] Handling HTTP request for try.com:80
[2024-12-08 14:10:15] Connected to target server: try.com:80
[2024-12-08 14:10:15] Forwarded request to target server: try.com:80
[2024-12-08 14:10:15] Received headers: HTTP/1.1 308 Permanent Redirect

Alt-Svc: h3=":443"; ma=2592000

Location: https://try.com/

Server: Framer/32b700c

Strict-Transport-Security: max-age=31536000

Date: Sun, 08 Dec 2024 12:10:15 GMT

Content-Length: 0
[2024-12-08 14:10:15] Cached response for http://try.com/. Expires at: 1733659875.6642213
[2024-12-08 14:10:15] Cache entry added to database for http://try.com/
```

The command was successful, indicating that the blacklist is working correctly.

6.6 - Testing whitelist:

After executing curl.exe -x http://127.0.0.1:9090 http://unlisted.com with unlisted.com being not present on either the whitelist or the blacklist, the send was forbidden, and the log file indicated the following:

```
[2024-12-08 14:27:25] Accepted connection from ('127.0.0.1', 51351)
[2024-12-08 14:27:25] New connection. Active connections: 0
[2024-12-08 14:27:25] Client connected: 127.0.0.1:51351
[2024-12-08 14:27:25] Request parsed: Method=GET, URL=http://unlisted.com/, Host=unlisted.com, Port=80
[2024-12-08 14:27:25] Request blocked: unlisted.com is not in the whitelist
[2024-12-08 14:27:25] Blocked request to unlisted.com due to whitelist/blacklist restrictions.
[2024-12-08 14:27:25] Sent 403 Forbidden response to client
[2024-12-08 14:27:25] Client connection closed after rejection.
[2024-12-08 14:27:25] Connection closed. Active connections: -1
```

After adding unlisted.com to the whitelist, the send was successful as expected:

```
[2024-12-08 14:32:07] Accepted connection from ('127.0.0.1', 51550)
[2024-12-08 14:32:07] New connection. Active connections: 0
[2024-12-08 14:32:07] Client connected: 127.0.0.1:51550
[2024-12-08 14:32:07] Request parsed: Method=GET, URL=http://unlisted.com/, Host=unlisted.com, Port=80
[2024-12-08 14:32:07] Request allowed: unlisted.com
[2024-12-08 14:32:07] Handling HTTP request for unlisted.com:80
[2024-12-08 14:32:07] Connected to target server: unlisted.com:80
[2024-12-08 14:32:07] Forwarded request to target server: unlisted.com:80
[2024-12-08 14:32:08] Received headers: HTTP/1.1 301 Moved Permanently

Date: Sun, 08 Dec 2024 12:32:07 GMT

Location: http://kennethcole.com

Content-Length: 0
[2024-12-08 14:32:08] Cached response for http://unlisted.com/. Expires at: 1733661188.1820042
[2024-12-08 14:32:08] Cache entry added to database for http://unlisted.com/
```

Note: If the URL was in both the blacklist and whitelist, the response will be false, and the log will return that the host is in both the blacklist and whitelist.

6.7 - Testing HTTPS:

We curled google.com after adding google.com to the whitelist:

```
[2024-12-08 15:23:17] Accepted connection from ('127.0.0.1', 52261)
[2024-12-08 15:23:17] New connection. Active connections: 0
[2024-12-08 15:23:17] Client connected: 127.0.0.1:52261
[2024-12-08 15:23:17] Request parsed: Method=CONNECT, URL=www.google.com:443, Host=www.google.com, Port=443
[2024-12-08 15:23:17] Request allowed: www.google.com
[2024-12-08 15:23:17] Handling HTTPS CONNECT request for www.google.com:443
[2024-12-08 15:23:17] Established secure tunnel to target server: www.google.com:443
[2024-12-08 15:23:17] Sent 200 Connection Established to client
[2024-12-08 15:23:18] Client closed the connection.
[2024-12-08 15:23:18] Client socket closed.
[2024-12-08 15:23:18] Target socket closed.
[2024-12-08 15:23:18] Encrypted traffic relayed between client and target server: www.google.com:443
[2024-12-08 15:23:18] Connection closed. Active connections: -1
```

We clearly got a successful transfer.

6.8 - Testing cache cleanup:

After curling example.com for the first time, we executed

sleep 60

curl.exe -x http://127.0.0.1:9090 http://example.com

To make sure the cache has enough time to get cleaned and clean the expired version of example.com. The command returned that the entry is no longer in the cache and fetched it from the main host.

```
[2024-12-08 15:33:21] Cache entry expired and removed: http://example.com/
[2024-12-08 15:33:21] Expired cache entries removed from database.
[2024-12-08 15:33:30] Accepted connection from ('127.0.0.1', 52432)
[2024-12-08 15:33:30] New connection. Active connections: 0
[2024-12-08 15:33:30] Client connected: 127.0.0.1:52432
[2024-12-08 15:33:30] Request parsed: Method=GET, URL=http://example.com/, Host=example.com, Port=80
[2024-12-08 15:33:30] Request allowed: example.com
[2024-12-08 15:33:30] Handling HTTP request for example.com:80
[2024-12-08 15:33:30] Connected to target server: example.com:80
[2024-12-08 15:33:30] Forwarded request to target server: example.com:80
[2024-12-08 15:33:30] Received headers: HTTP/1.1 200 OK

Accept-Ranges: bytes

Age: 240597

Cache-Control: max-age=604800

Content-Type: text/html; charset=UTF-8

Date: Sun, 08 Dec 2024 13:33:29 GMT

Etag: "3147526947"

Expires: Sun, 15 Dec 2024 13:33:29 GMT

Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT

Server: ECAcc (dcd/7D7F)
```

Testing with apache

```
rebecca@DESKTOP-JM4EBSO:/mnt/c/Users/Hp/Desktop/UNI/Fall 2024/network/project/project$ ab -n 1000 -c 10 -X 127.0.0.1:9090 http://example.com/
This is ApacheBench, Version 2.3 <$Revision: 1879490 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking example.com [through 127.0.0.1:9090] (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests


Server Software:        ECAcc
Server Hostname:        example.com
Server Port:            80

Document Path:          /
Document Length:        1256 bytes

Concurrency Level:      10
Time taken for tests:   85.770 seconds
Complete requests:      1000
Failed requests:        3
   (Connect: 0, Receive: 0, Length: 3, Exceptions: 0)
Total transferred:      1632347 bytes
HTML transferred:       1255437 bytes
Requests per second:    11.66 [#/sec] (mean)
Time per request:       857.698 [ms] (mean)
Time per request:       85.770 [ms] (mean, across all concurrent requests)
Transfer rate:          18.59 [Kbytes/sec] received
```

```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    2  48.3      0    1080
Processing:   113  845 147.6    833    2247
Waiting:      113  842 130.3    833    1761
Total:        113  847 153.0    833    2247

Percentage of the requests served within a certain time (ms)
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    2  48.3      0    1080
Processing:   113  845 147.6    833    2247
Waiting:      113  842 130.3    833    1761
Total:        113  847 153.0    833    2247

Percentage of the requests served within a certain time (ms)
  50%    833
  66%    877
  75%    905
  80%    923
  90%    973
  95%   1020
  98%   1113
  99%   1343
 100%   2247 (longest request)
```

Phase	Mean (ms)	Max (ms)	Analysis
Connect:	2 ms	1080 ms	Most connections were established quickly; a few outliers caused delays.
Processing:	845 ms	2247 ms	Response times were consistent, but some outliers caused higher response times.
Waiting:	842 ms	1761 ms	Indicates most time was spent waiting for the server's response, suggesting server latency.

1. Summary of the Test

- Requests Sent: 1000
- Concurrency Level: 10 (10 simultaneous connections)
- Total Time Taken: 86.770 seconds
- Requests per Second: 11.66 [#/sec] (mean)
- Document Length: 1256 bytes (size of each response)
- Transfer Rate: 18.59 KB/sec
- The server handled 1000 requests with an average throughput of 11.66 requests per second.
- The concurrency level of 10 ensured simultaneous requests were processed.
- The transfer rate of 18.59 KB/sec indicates that the server and proxy are handling moderate-sized responses efficiently.

2. Failed Requests

- Failed Requests: **3**
    - Length Mismatch: 3 (indicates the content length in the response did not match the expected length).
- There were no connection or receive errors, meaning the proxy and target server were reachable and stable.
- Length mismatches could indicate issues with partial responses or inconsistencies between content-length headers and actual response content.

3. Connection Times (ms)

- Min: 113 ms
- Mean: 847 ms
- Median: 833 ms
- Max: 2247 ms

4. Percentiles (Request Completion Time)

- 50% of requests: Completed within 833 ms.
- 75% of requests: Completed within 905 ms.
- 90% of requests: Completed within 973 ms.
- 99% of requests: Completed within 1343 ms.
- 100% of requests: Longest request took 2247 ms.
- Most requests were handled within ~1 second, with 99% completed within 1.3 seconds.
- The longest request (2247 ms) is an outlier and may be due to network latency or server load.

Performance Observations

1. Good Average Throughput:

With 11.66 requests per second and 10 concurrent connections, the proxy is reasonably responsive under this load.

2. Stable Proxy Server:

The proxy server handled the test well with only a few failed requests, none related to connection issues.

3. Server Latency:

The majority of the time was spent waiting for the target server (example.com), which contributed to the overall high response times. We assume those are the non-cached requests.

6 - Work Distribution

- The Main Proxy functionalities were worked on by Abdulrahman, Ikram and Rebecca.
- The HTTPS part was worked on by Abdulrahman.
- The admin interface was worked on by Abdulrahman and Ikram.
- The testing part was worked on by Rebecca.
- The document was worked on by all 3.

7 - Full List of Features in Bullet Points (so that Dr. Ayman does not get tired)

Core Proxy Functionalities:

1. **Request Parsing:**
   - Extracts the HTTP method, URL, host, and port from client requests.
   - Handles CONNECT requests for HTTPS separately.

2. **Header Modification:**
   - Updates the Host header to match the target server.
   - Removes Proxy-Connection headers to comply with forwarding requirements.

3. **HTTP Request Handling:**
   - Establishes connections to the target server for HTTP requests.
   - Forwards the client's requests after parsing and modifying headers.
   - Relays server responses back to the client.

4. **HTTPS Tunneling:**
   - Handles CONNECT requests by creating a secure tunnel using SSL.
   - Forwards encrypted data between the client and the server.

5. **Data Forwarding:**
   - Uses non-blocking I/O with select to relay data efficiently between the client and the target server.

Advanced Features:

6. **Caching:**
   - Caches responses from target servers to reduce latency for repeated requests.
   - Implements cache expiration based on Cache-Control and Expires headers.
   - Provides in-memory caching with SQLite database support for persistence.

7. **Cache Cleanup:**
   - Periodic cleanup of expired cache entries to maintain efficiency.
   - Logs cache states for transparency.

8. **Whitelist and Blacklist Management:**
   - Dynamically loads and refreshes whitelist/blacklist entries from a SQLite database.
   - Blocks or allows requests based on the domain's presence in these lists.

9. **Admin Interface (via Flask):**
   - Login/logout functionality for admin users.
   - Web-based management of the blacklist, whitelist, and cache entries.
   - View logs and monitor active connections through an admin dashboard.

10. **Logging:**
    - Logs request handling, connection statuses, and errors.
    - Maintains a log file for debugging and operational insights.

11. **Active Connection Tracking:**
    - Tracks the number of active connections.
    - Updates the count dynamically in the SQLite database and admin interface.

Extra Functionalities:

12. **Scheduled Cache Cleanup:**
    - Automates the cleanup process at regular intervals.
    - Ensures expired entries are removed without manual intervention.

13. **Dynamic Resource Refreshing:**
    - Periodically refreshes the blacklist and whitelist to reflect database changes.
    - Supports real-time updates without restarting the proxy server.

14. **SQLite Database Integration:**
    - Manages all configurations (e.g., blacklist, whitelist, cache, and active connections) using a database backend.

15. **Custom Error Handling:**
    - Sends user-friendly error responses (e.g., 403 Forbidden, 504 Gateway Timeout) to clients when issues occur.

16. **Secure SSL Connections:**
    - Ensures end-to-end encryption for HTTPS traffic by implementing secure sockets.

17. **Admin Features:**
    - Allows admins to monitor and manage active connections, logs, cache, blacklist, and whitelist through an interface.

18. **UI Templates:**
    - User-friendly web pages for blacklist/whitelist management, cache visualization, and logs review.

8 - Sources

- The following video was used for practice: https://youtu.be/3QiPPX-KeSc?si=tMh93Jb8Z3nRnJUg
- ChatGPT helped with logging and debugging.